Today

• Deadlocks
• Detection algorithm
Racing for resources

• Threads are racing to acquire resources.
  – Threads may belong to different processes.
  – Resources may be logical (user data, OS structures) or physical (memory, printer, disk).

• Assume there is a mechanism that coordinates the access of threads to resources.

• This mechanism may be a combination of:
  – Synchronization primitives.
  – The operating system.
  – Resources themselves.
Safety property

- Coordinating threads involves blocking threads until resources are available.
- This coordination mechanism should satisfy the safety property: deadlock freedom!
  - At any point of time, at least one thread should be able to make progress.
- Undesirable scenario:
  - Process A acquires resource 1, and is waiting for resource 2
  - Process B acquires resource 2, and is waiting for resource 1
  - Deadlock!
Example 1: Semaphores

semaphore: mutex1 = 1 /* protects file */
mutex2 = 1 /* protects printer */

Process A code:
{ /* initial compute */
  P(mutex1)
P(mutex2)
/* use file & printer*/
V(mutex2)
V(mutex1)
}

Process B code:
{ /* initial compute */
P(mutex2)
P(mutex1)
/* use file & printer */
V(mutex1)
V(mutex2)
}
Example 2: Dining Philosophers

class Philosopher:
    chopsticks[N] = [Semaphore(1),…]

    Def __init__(mynum):
        self.id = mynum

    Def eat():
        right = (self.id+1) % N
        left = (self.id-1+N) % N
        while True:
            P(left)
            P(right)
            # eat
            V(right)
            V(left)
Deadlock

• A cycle of waiting among a set of threads where each thread is waiting for some other thread in the cycle to take some action.
• Caused by the coordination mechanism.
Four Conditions for Deadlock

- **Mutual Exclusion**
  - At least one resource must be held in non-sharable mode.

- **Hold and wait**
  - There exists a process holding a resource, and waiting for another.

- **No preemption**
  - Resources cannot be preempted.

- **Circular wait**
  - There exists a set of processes \( \{P_1, P_2, \ldots, P_N\} \), such that
    - \( P_1 \) is waiting for \( P_2 \), \( P_2 \) for \( P_3 \), \ldots and \( P_N \) for \( P_1 \).

- If some of these conditions do not hold, then there is no deadlock (necessary conditions).

- If all four conditions hold, then there may not be a deadlock (not sufficient conditions).
Deadlock Detection

• Stop the world.
• Check if the conditions for which threads are waiting can be ever satisfied.
  – Check if requested resources can ever be allocated to threads.
Resource Allocation Graph (RAG)

• 2 kinds of nodes
• A process $P_3$ represented as:
• A resource $R_7$ represented as:
  – A resource often has multiple identical units, such as “blocks of memory”.
  – Represent these as circles in the box.
• Edge from $P_3$ to $R_8$:
  – $P_3$ wants $k$ units from $R_8$.
• Edge from $R_5$ to $P_6$:
  – $P_6$ has $m$ units from $R_5$. 
Can all requests be satisfied?
Deadlock detection with RAG

Start satisfying the requests of each process, until:

• no process is left → no deadlock, or
• no remaining request can be satisfied → deadlock.
RAG reduction

• Find satisfiable process P:
  – available amount of resource \( \geq \) amount requested.
• Erase P.
  – Intuition: Grant the request, let it run, eventually it will release the resource.
• Repeat until all processes gone or irreducible.
Is this graph reducible?

Yes! The system is not deadlocked.
Is this graph reducible?

No! The system is deadlocked.
Detection Algorithm

Data structures:

- \( n \): number of processes
- \( m \): number of resource types
- \( \text{available}[1..m] \): \( \text{available}[j] \) is number of available resources of type \( j \)
- \( \text{request}[1..n,1..m] \): current demand of each \( P_i \) for each \( R_j \)
- \( \text{allocation}[1..n,1..m] \): current allocation of resource \( R_j \) to \( P_i \)
- \( \text{free}[1..m] \): \( \text{free}[j] \) is number of free resources of type \( j \) (not used by any process)
- \( \text{finish}[1..n] \): true if \( P_i \)'s request can be satisfied
Detection Algorithm

1. free[] = available[]
2. for all processes i: finish[i] = allocation[i] == [0,0,...,0])
3. find a process i such that finish[i]=false and request[i] <= free
   if no such i exists, goto 7
4. free = free + allocation[i]
5. finish[i]=true
6. goto 3
7. system is deadlocked iff finish[i]=false for some process i
Detection Algorithm: Example

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 R2 R3</td>
<td>R1 R2 R3</td>
<td>R1 R2 R3</td>
</tr>
<tr>
<td>P0 0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P1 2 0 0</td>
<td>2 0 2</td>
<td></td>
</tr>
<tr>
<td>P2 3 0 3</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>P3 2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>P4 0 0 2</td>
<td>0 0 2</td>
<td></td>
</tr>
</tbody>
</table>

- The system is not in a deadlocked state.
- What will happen if P2 makes an additional request for one instance of type R3?
Dealing with Deadlocks

Reactive Approaches:

- Periodically check for evidence of deadlock
  - For example, using a graph reduction algorithm
- Then need a way to recover
  - Could blue screen and reboot the computer
  - Could pick a “victim” and terminate that thread
    - But this is only possible in certain kinds of applications
    - Basically, thread needs a way to clean up if it gets terminated and has to exit in a hurry!
  - Often thread would then “retry” from scratch

(despite drawbacks, database systems do this)
Dealing with Deadlocks

Proactive Approaches:

- Deadlock Prevention and Avoidance
  - Prevent one of the 4 necessary conditions from arising
  - .... This will prevent deadlock from occurring
Today

- Deadlocks
- Detection algorithm
Coming up...

• Next lecture: prevention and avoidance of deadlocks
• HW2: due tonight
• In-class exam: tomorrow, last 30mins